

06/02/99
JCS53 U.S. PTO

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Inventorship.....Bittner et al.
Applicant.....Microsoft Corporation
Attorney's Docket No.MS1-317US
Title: Dynamic Address Windowing On A PCI Bus

TRANSMITTAL LETTER AND CERTIFICATE OF MAILING

To: Commissioner of Patents and Trademarks,
Washington, D.C. 20231

From: James R. Banowsky (Tel. 509-324-9256; Fax 509-323-8979)
Lee & Hayes, PLLC
421 W. Riverside Avenue, Suite 500
Spokane, WA 99201

The following enumerated items accompany this transmittal letter and are being submitted for the matter identified in the above caption.

1. Specification—title page, plus 25 pages, including 32 claims and Abstract
2. Transmittal letter including Certificate of Express Mailing
3. 4 Sheets Formal Drawings (Figs. 1-6)
4. Return Post Card

Large Entity Status [x]

Small Entity Status []

Date: 6-2-99

By: James R. Banowsky
James R. Banowsky
Reg. No. 37,773

CERTIFICATE OF MAILING

I hereby certify that the items listed above as enclosed are being deposited with the U.S. Postal Service as either first class mail, or Express Mail if the blank for Express Mail No. is completed below, in an envelope addressed to The Commissioner of Patents and Trademarks, Washington, D.C. 20231, on the below-indicated date. Any Express Mail No. has also been marked on the listed items.

Express Mail No. (if applicable) _____

EL319764493

Date: June 2, 1999

By: Helen M. Hare
Helen M. Hare

EL319764493

JCS53 U.S. PTO
06/02/99

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

Dynamic Address Windowing On A PCI Bus

Inventor(s):

Ray A. Bittner

Michael Ginsburg

ATTORNEY'S DOCKET NO. MS1-317US

09324905-00000000

TECHNICAL FIELD

This invention relates to the field of digital computer systems, and more particularly, to the addressing of PCI devices in a general-purpose multi-tasking operating system having a CPU which provides a number of address lines less than that required by a PCI device bus.

BACKGROUND

Digital computer operating systems periodically schedule processes for the central processing unit (CPU) of the computer to perform. The operating system schedules sub-parts of executable applications, known as threads, on the CPU. In multi-tasking operating systems, threads can be independently scheduled for execution on the processor so that more than one computer program or application may proceed at the same time. While the CPU only works on one thread at a time, it appears to the user that the CPU is running multiple applications concurrently. During multi-tasking, the CPU interrupts execution of one thread to begin execution of another thread or to resume execution of a previously interrupted thread. Switching from one thread to another is known as a "context switch."

One multi-tasking operating system currently available is the "WINDOWS CE" operating system developed by Microsoft Corporation. "WINDOWS CE" supports a wide variety of CPUs. Many of these CPUs do not provide full 32-bit addressing on the external pins of the device. This presents a problem in that many of today's computer systems utilize fast I/O devices with high-speed 32-bit buses, such as the industry standard peripheral component interconnect (PCI) bus. PCI devices, such as video controllers, are assigned a range of contiguous addresses on the bus through which they may be accessed.

1 To enable access to several PCI devices, each PCI device is assigned a
2 unique, non-overlapping address range, whereby the high order bits of the 32-bit
3 address are used to distinguish among address spaces for different devices. For
4 instance, the upper two bits of a 28-bit address might be used to differentiate
5 between four different PCI devices, with the remaining 26 bits being used for
6 addressing within the devices themselves. Unfortunately, the CPU itself may not
7 support 28 bits, but only 26 bits. Accordingly, there is no way for the CPU to
8 distinguish amongst the different PCI devices on the bus. When a CPU is
9 incapable of providing the full 32-bit PCI address space, there is a need to extend
10 the addressing output of the CPU to properly access a PCI device connected to a
11 PCI bus.

12 Special chips, referred to as "PCI host bridges," are available to interface a
13 CPU to a PCI bus. Some of these chips have circuitry for extending the limited
14 address range of CPUs such as those discussed above.

15 Fig. 1 shows a system utilizing a PCI host bridge. The system includes a
16 prior art computer 10 having a microprocessor 12 coupled to a PCI host bridge 14.
17 The microprocessor 12 has a central processing unit (CPU) 16. The CPU 14 is
18 connected to multiple devices, such as a PCMCIA driver 18, a universal serial bus
19 (USB) 20, an I/O (input/output) port 22, and a bus interface unit (BIU) 24, via an
20 address bus 26. The BUI 24 interfaces the microprocessor 12 to the PCI host
21 bridge 14 via a standard address bus 30 having less than thirty-two lines. The PCI
22 host bridge 14 converts addresses received from the address bus 30 to full 32-bit
23 addresses output over a 32-bit PCI bus 32. One or more PCI devices are coupled
24 to the PCI bus 32, as represented by devices 34a, 34b, and 34c.
25

The PCI host bridge 14 has an extension register 36 to extend the address on the bus 32 (i.e., less than 32 bits) to a full 32-bit address suitable for the PCI bus 32. More particularly, the extension register 36 contains memory to store a number of bits which are concatenated with lower-order bits from the standard address bus 30 to produce a full 32-bit address for one of the PCI devices 34.

The extension register 36 is read from and written to by the CPU 16 to allow the CPU to set the value of the high order address bits. By changing the values in the extension register, the CPU 16 is able to address the full address space supported by the PCI bus 32. For example, assume that a CPU 16 outputs a 26-bit address on bus 30. Further assume that one of the PCI devices 34 is a video device that has been assigned an address range of A4000000 through A7FFFFFF (expressed in hexadecimal notation). To address the video device, the CPU 16 writes the binary value 101001 to the extension register 36 to set the upper six bits of the PCI address. The external address pins of the CPU 16 then supply the low order twenty-six bits during normal bus cycle generation. If the CPU is then required to address a different PCI device, the CPU 16 changes the value in the extension register 36 to match the high order address bits of the target PCI device. For example, if the second PCI device is mapped to the address range of A8000000 through ABFFFFFF, the CPU 16 writes binary value 101010 to the extension register 36 and supplies the lower order twenty-six bits of the address during the bus cycle generation.

The procedure operates adequately within a non-multi-tasking environment, but not in a multi-tasking operating system environment. In present multi-tasking systems, the operating systems are designed to work with specific CPUs that only supply an adequate number of address lines, thereby avoiding the problem

1 described above. The operating system may update the extension register once at
2 system initialization and never update it again since the address bits supplied by
3 the extension register are common across all PCI devices. This situation does not
4 pose any problems because the correct address is generated while the PCI device
5 is accessed.

6 However, consider the situation of a multi-tasking operating system
7 working with CPUs that do not provide a sufficient number of address lines. The
8 process described must be maintained even though the operating system may
9 suspend or resume execution of a particular thread at any given time. This poses
10 some potentially troublesome problems in that the extension register may contain
11 the wrong values as a result of the context switching performed by the multi-
12 tasking operating system.

13 Suppose, for example, two threads A and B are accessing PCI devices 1
14 and 2, respectively, via the PCI host bridge. Thread A needs the extension register
15 set to a value "X" to access PCI Device 1 and thread B requires the extension
16 register to have a value "Y" to access PCI Device 2. Table 1 describes the
17 situation.

Table 1: Inter-process Conflict In A Multi-tasking Operating System

Time	Thread A	Thread B	Extension Register Value
T0	Extension Register = X		X
T1	Access PCI Device 1		X
T2	Context Switch		X
T3		Extension Register = Y	Y
T4		Access PCI Device 2	Y
T5	Context Switch		Y
T6	Access PCI Device 1		Y

At time T0, thread A requests the CPU to set the extension register to value X. The CPU then accesses PCI device 1 at time T1. At time T2, the operating system performs a context switch and schedules thread B for execution. At time T3, thread B requests the CPU to set the extension register to value Y. The CPU then accesses PCI device 2 at time T4.

Thereafter, the operating system performs another context switch at time T5 to return to thread A for more execution. However, note that the extension register value remains at the value Y (the value for thread B) rather than the value X for thread A. The value in the extension register is not updated at this time because such an update only occurs when a thread – in this case, thread A – is initially called. As a result, when thread A attempts to access PCI device 1 again at time T6, the extension register contains value Y (i.e., the extension value for PCI device 2) and the access attempt fails.

One way to resolve this problem is to disable the context switching in the operating system while a PCI device is being accessed. This, however, could compromise the performance of the operating system since it may result in “jerky” operation of the applications being performed and would adversely affect the

1 illusion of multi-tasking to the user. Additionally, in time-critical, real-time
2 operating systems that are used in critical medical equipment or aircraft flight
3 control systems, it is absolutely necessary to guarantee that context switches will
4 occur within a specific time period. If context switching is disabled for an
5 arbitrary period of time while a PCI device is being accessed in such a system, the
6 time-critical thread may not execute within its allotted time period, thereby
7 causing failure of the critical system.

8 A better solution to the problem is to provide a way for the operating
9 system to restore the value of the extension register prior to each context switch so
10 that the correct PCI device is accessed by the operating system every time. This
11 solution is accomplished by the present invention.

12 13 SUMMARY

14 The present invention concerns a multi-tasking operating system and
15 method that updates extension register values to ensure that various threads utilize
16 the correct values to access corresponding PCI devices. In this manner, when
17 execution of an interrupted thread is resumed, the thread will be able to access the
18 correct PCI device.

19 When a first thread of an application program requires access to a PCI
20 device, the operating system writes the high order bits of the PCI device address to
21 two places: (1) the extension register of the PCI host bridge and (2) a separate
22 memory location associated with the first thread. Similarly, when a second thread
23 is started (i.e., the first thread is suspended) and desires access to a second PCI
24 device, the operating system writes the high order bits of the second PCI device
25 address to two places: (1) the extension register of the PCI host bridge and (2) a

1 second memory location associated with the second thread. In the described
2 implementation, the separate memory locations are implemented as a table that
3 holds the upper address bits of a PCI device addresses in correlation with the
4 threads that call the devices.

5 When a subsequent context switch occurs from the second thread back to
6 the first thread (i.e., the second thread is suspended and execution of the first
7 thread is resumed), the operating system retrieves the stored value from the
8 memory location associated with the first thread and writes the value to the
9 extension register. Therefore, when the first thread tries to access its particular
10 PCI device, the proper value is already located in the extension register and can be
11 concatenated with the CPU output address to access the PCI device.

12 When the next context switch to the second thread occurs, the operating
13 system retrieves the stored value from the memory location associated with the
14 second thread and writes the value to the extension register. Therefore, the proper
15 value is located in the extension register so that the correct PCI device is accessed
16 by the second thread.

17 18 **BRIEF DESCRIPTION OF THE DRAWINGS**

19 Fig. 1 is a block diagram of a prior art computer having a microprocessor
20 interfacing with a PCI host bridge to provide a PCI address to multiple PCI
21 devices via a PCI bus.

22 Fig. 2 is a block diagram of a computer having a multi-tasking operating
23 system with a thread scheduler for scheduling threads in conjunction with a PCI
24 host bridge to enable access to PCI devices on a PCI bus.
25

1 Fig. 3 is a block diagram of the multi-tasking operating system, which
2 shows an extension register initialization function and an extension register value
3 table used in scheduling threads.

4 Figs. 4-6 is a flow diagram of a process for managing the extension register
5 to update values therein with each context switch among multiple threads. Fig. 4
6 shows steps implemented by an application program (or its corresponding thread
7 and driver). Fig. 5 shows steps performed by a multi-tasking operating system.
8 Fig. 6 shows steps performed by the extension register initialization function.

9 10 **DETAILED DESCRIPTION**

11 The present invention concerns a multi-tasking operating system and
12 methods that enable extension of reduced-size addresses output by a CPU to full-
13 size addresses carried over a PCI bus for accessing various PCI devices. The
14 multi-tasking operating system may be implemented in a number of computing
15 devices. For discussion purposes, the operating system is described in the context
16 of a handheld computer.

17 Fig 2 is a block diagram showing pertinent functional components of a
18 computer 40 in which the present invention may be implemented. The computer
19 40 is similarly constructed as computer 10 in Fig. 1 with respect to microprocessor
20 12, PCI host bridge 14, and PCI devices 34a-34c. The microprocessor 12 outputs
21 reduced-sized addresses (e.g., less than 32 bits) on bus 30, and the PCI host bus 14
22 extends the addresses to full-size 32-bit addresses for the PCI bus 32.

23 The computer 40 has a nonvolatile memory 50 that stores a multi-tasking
24 operating system 52 with a thread scheduler 54. The multi-tasking operating
25 system 52 supports concurrent operation of multiple application programs, as

1 represented by four applications 56a, 56b, 56c, and 56d (although more or less
2 applications may be used). The microprocessor 12 communicates with the
3 memory 50 via a standard data bus 58, which is well known in the art.

4 The multi-tasking operating system 52 assigns threads for associated
5 applications 56a, 56b, 56c, and 56d. As noted in the Background, one problem
6 encountered in the context of multi-tasking operating systems executing on CPUs
7 with less than the number of address pins for full bus addressing occurs during
8 context switching among the multiple threads. A context switch occurs whenever
9 the multi-tasking operating system interrupts one thread to execute another thread.
10 The problem is that, following a context switch, the extension register 36 may not
11 contain the appropriate value for the current thread to access a PCI device on the
12 PCI bus, but instead might contain a value used by a previously executed thread.

13 To overcome this problem, the multi-tasking operating system 52 stores the
14 addressing values used in the extension register 36 for each thread and updates the
15 extension register 36 each time the multi-tasking operating system performs a
16 context switch between two threads. More particularly, when thread A of
17 application program 56a requires access to a PCI device 34a, the operating system
18 writes the high order bits of the PCI device address to two places: (1) the
19 extension register 36 of the PCI host bridge 14 and (2) a memory location
20 associated with thread A. Similarly, when thread B is subsequently started (i.e.,
21 thread A is suspended) and desires access to a second PCI device, the operating
22 system writes the high order bits of the second PCI device address to two places:
23 (1) the extension register 36 of the PCI host bridge 14 and (2) a second memory
24 location associated with the second thread.
25

1 When a context switch occurs from thread B back to thread A, the
2 operating system retrieves the stored value from the memory location associated
3 with thread A and writes the value to the extension register 36. Therefore, when
4 thread A tries to access its particular PCI device, the proper address value is
5 already located in the extension register and can be concatenated with the CPU
6 output address to access the PCI device.

7 Fig. 3 shows the memory 50 and components of the multi-tasking operating
8 system 52 in more detail. In addition to thread scheduler 54, the operating system
9 52 includes an extension register value table 60 and an extension register
10 initialization function 62. The extension register value table 60 stores unique
11 extension values used to address the PCI devices in association with the
12 application programs that are accessing the PCI devices. In the illustrated
13 example, the table 60 stores extension registration values 64a-64d for the four
14 threads A, B, C, and D associated with application programs 56a-56d. The
15 operating system also includes one or more device drivers 66.

16 Figs. 4-6 show a general process for managing the extension register to
17 update values therein with each context switch among multiple threads. Figs. 4
18 and 5 illustrate an initializing process when an application thread decides to write
19 to a peripheral device for the first time. Fig. 6 illustrates a process in which the
20 operating system switches contexts among existing threads. The processes are
21 described with additional reference to Figs. 2 and 3. The processes are performed
22 in software by code embodied in various software components stored in memory,
23 including an application program 56 (or corresponding OS-based thread) (Fig. 4),
24 the extension register initialization function 62 (Fig. 5), and the operating system
25 52 (Fig. 6).

1 With reference to Fig. 4, an application program 56a prepares to write data
2 to a peripheral device 34a via the PCI host bridge 14 and PCI bus 32. The
3 application program 56a is executed by a corresponding thread A, which in turn
4 executes under the control and supervision of operating system 52. Prior to
5 utilizing peripheral device 34a, the thread A calls the extension register
6 initialization function 62 and passes in a base address for use by the extension
7 register for properly addressing the peripheral device 34a (step 100 in Fig. 4). The
8 base address might be a full address from which an address extension is
9 calculated, or only the upper bits that form the desired address extension. The
10 initialization function then performs several steps, as discussed below with respect
11 to Fig. 5. Upon completion by the initialization function 62, thread A writes data
12 to or reads data from the peripheral device 34a (step 102 in Fig. 6).

13 The actual access of PCI devices is often accomplished through the use of
14 operating system device drivers such as drivers 66 shown in Fig. 3. In these
15 situations, the call to register initialization function 62, and the subsequent access
16 to the PCI device itself, is made by the device driver. Each device driver is
17 designed to call register initialization function 62 with a desired base address or
18 address extension prior to accessing a PCI device.

19 With reference to Fig. 5, when the extension register initialization function
20 62 is called (step 100 in Fig. 4), the function 62 receives a base address or address
21 extension value for accessing the PCI peripheral device 34a from the thread A
22 (step 110 in Fig. 5). The initialization function 62 then writes the appropriate
23 extension register value to the extension register 36 in PCI host bridge 14 (step
24 112 in Fig. 5). In the case where register initialization function 62 receives a base
25 address rather than the extension value itself, function 62 calculates the

1 appropriate extension value—in most cases by determining the value of the
2 uppermost bits of the base address.

3 Contemporaneously with this write operation (step 112), the extension
4 register initialization function 62 writes the received extension register value or
5 base address to a memory location and associates the value with the thread A (step
6 114 in Fig. 5). In the described implementation, the value is stored in the
7 extension register value table 60 as entry 64a. Although a table is illustrated, it
8 should be understood that other techniques may be used to store the extension
9 value in association with the calling thread.

10 After the extension register initialization function 62 finishes, control is
11 then returned to the application program thread A to write data to or read data
12 from the peripheral device 34a.

13 Now, suppose a second application 56b is launched. The thread scheduler
14 54 of multi-tasking operating system 52 starts another thread B to execute the
15 newly launched application program 56b. In addition, suppose the second
16 application program 56b wants to write data to a second peripheral device 34b
17 over the PCI bus 32. For this to occur, the extension register 36 in PCI host bridge
18 14 must be updated if and when the thread A currently executing is interrupted by
19 the thread scheduler 54 to execute the second thread B of the second application
20 program 56b.

21 Prior to any attempt to access the second peripheral device 34b, the thread
22 B calls the extension register initialization function 62 and passes in a base address
23 or extension value for use in properly addressing the peripheral device 34b (i.e.,
24 step 100 in Fig. 4). The extension register initialization function 62 writes the
25

1 received or calculated value to the extension register 36 and to the extension
2 register value table 60 as entry 64b (i.e., steps 112 and 114 in Fig. 5).

3 With multiple threads executing, the multi-tasking operating system 52
4 performs context switches between the threads A and B. The operating system 52
5 executes thread B of the application program 56b until it is completed or until it is
6 interrupted by the thread scheduler 54 to perform a context switch to thread A of
7 the first application program 56a.

8 Fig. 6 shows a process for updating the extension register in response to
9 context switching performed by the operating system. For discussion purposes,
10 the following example assumes that thread B is currently executing and that the
11 thread next scheduled for execution is thread A of the first application program
12 56a.

13 When the multi-tasking operating system 52 performs a context switch
14 from thread B back to thread A, the operating system 52 retrieves the extension
15 register value for thread A from entry 64a of extension register table 60 (step 120
16 in Fig. 6). The operating system then writes the extension register value to the
17 address extension register 36 so that the thread A will be able to access the first
18 PCI device 34a (step 122 in Fig. 6). The multi-tasking operating system 52 then
19 resumes execution of the thread of the application program 50a (step 124 in Fig.
20 6).

21 This context switching process of Fig. 6 may be repeated as desired for
22 however many threads are being executed.

23 According to another aspect of this invention, a default extension value is
24 utilized for any threads that have not called register initialization function 62.
25 When a thread is initiated, its corresponding entry in table 60 is initialized with

1 this default value. This default value can be overridden by any particular thread,
2 by calling register initialization function 62 prior to accessing a PCI device.

3 Although the invention has been described in language specific to structural
4 features and/or methodological steps, it is to be understood that the invention
5 defined in the appended claims is not necessarily limited to the specific features or
6 steps described. Rather, the specific features and steps are disclosed as preferred
7 forms of implementing the claimed invention.

8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

1 **CLAIMS**

2 1. In a computing device having a processor that generates a first
3 address signal of a first width and one or more peripheral devices that are
4 addressed with a second address signal of a second width that is greater than the
5 first width, wherein the second address signal is produced in the computing device
6 by concatenating an address extension from an address extension register with the
7 first address signal, a method comprising:

8 concurrently executing threads of a plurality of application programs,
9 wherein different ones of the threads indicate one or more address extensions to an
10 operating system;

11 storing the address extensions for use by the operating system;

12 repeatedly switching between execution of the threads; and

13 prior to executing a particular thread, writing the address extension of the
14 base address indicated by the particular thread to the extension register.

15
16 2. A method as recited in claim 1, wherein the address extensions are
17 indicated as a value of the second width.

18
19 3. A method as recited in claim 1, wherein individual address extensions
20 identify address ranges associated with one or more peripheral devices.

21
22 4. A method as recited in claim 1, further comprising calling an
23 operating system device driver from one of the threads, wherein the device driver
24 invokes an initialization function to indicate the one or more base addresses.
25

1 5. A thread scheduler that schedules multiple execution threads for
2 interleaved execution by a processor, wherein the processor generates a processor
3 address signal that is combined with an extended address signal to create a
4 peripheral address signal, wherein the extended address signal is produced from a
5 value stored in an address extension register, and wherein the value is writeable
6 and readable by the processor; the thread scheduler performing steps comprising:

7 interrupting a first execution thread to execute portions of one or more
8 other execution threads;

9 recording the value from the address extension register;

10 restoring the recorded value to the extension register after executing said
11 portions of one or more other threads; and

12 resuming the first execution thread after restoring the recorded value to the
13 address extension register.

14
15 6. A thread scheduler as recited in claim 5, wherein the thread scheduler
16 records address extension register values associated with a plurality of interrupted
17 execution threads and restores the address extension register values to the address
18 extension register when resuming the associated execution threads.

19
20 7. One or more computer-readable storage media containing a program
21 that implements a thread scheduler as recited in claim 5.

22
23
24
25

1 **8.** A method of scheduling multiple execution threads for interleaved
2 execution by a processor, wherein the processor generates a processor address
3 signal, comprising:

4 executing a first execution thread;
5 writing an address extension to an extension register;
6 concatenating the address extension with the processor address signal to
7 create a peripheral address signal used by the first execution thread;
8 storing the address extension in a location other than the extension register;
9 interrupting the first execution thread to execute portions of one or more
10 other execution threads;
11 interrupting the one or more other execution threads to resume execution of
12 the first execution thread;
13 restoring the stored address extension to the extension register before
14 resuming executing of the first execution thread; and
15 resuming execution of the first execution thread.

16
17 **9.** A method as recited in claim 8, wherein each address extension
18 identifies an address range associated with one or more peripheral devices.
19

20 **10.** A method as recited in claim 8, further comprising:
21 writing a second address extension associated with the one or more other
22 execution threads to the extension register;
23 concatenating the second address extension with a second processor
24 address signal to create a second peripheral address signal used by the one or more
25 other execution threads; and

1 storing the second address extension in a second location other than the
2 extension register.

3
4 **11.** A computer-readable storage medium having instructions for
5 performing the steps recited in claim 8.

6
7 **12.** A multi-tasking operating system for use in a computing device
8 having a processor that generates a first address signal of a first width and one or
9 more peripheral devices that are addressed with a second address signal of a
10 second width that is greater than the first width, wherein the second address signal
11 is produced in the computing device by concatenating an address extension from
12 an address extension register with the first address signal, the operating system
13 being configured to perform steps comprising:

14 concurrently executing a plurality of application program threads;

15 storing address extensions corresponding to different ones of the
16 application program threads;

17 repeatedly switching between the application program threads; and

18 prior to executing any particular thread, writing the address extension
19 corresponding to that particular thread to the extension register.

20
21 **13.** A multi-tasking operating system as recited in claim 12, further
22 comprising:

23 a register initialization function that is callable by from the threads to
24 specify address extensions.

1 **14.** A multi-tasking operating system as recited in claim 12, wherein
2 each address extension identifies an address range associated with one or more
3 peripheral devices.

4
5 **15.** One or more computer-readable storage media containing a multi-
6 tasking operating system as recited in claim 12.

7
8 **16.** A computing device comprising:
9 a processor that generates a first address signal having a first width;
10 one or more peripheral devices that are addressed with a second address
11 signal having a second width that is greater than the first width;
12 an address extension register that stores an address extension, wherein the
13 address extension is combined with the first address signal to create the second
14 address signal; and
15 the processor being programmed to record the address extension being used
16 by a first execution thread, to interrupt the first execution thread to execute
17 portions of one or more other execution threads, to restore the recorded address
18 extension to the address extension register after executing said portions of one or
19 more other execution threads, and to resume the first execution thread after
20 restoring the recorded address extension to the address extension register.

21
22 **17.** A computing device as recited in claim 16 wherein each address
23 extension identifies an address range associated with one or more peripheral
24 devices.
25

1 **18.** A computing device as recited in claim 16 wherein the processor is
2 further programmed to record and restore address extension for a plurality of
3 execution threads.

4
5 **19.** A computing device as recited in claim 16 wherein the processor is
6 programmed to perform the following steps upon interrupting the first execution
7 thread to begin execution of the one or more other execution threads:

8 recording the address extension being used by the one or more other
9 execution threads; and

10 interrupting the one or more other execution threads to resume the first
11 execution thread.

12
13 **20.** A computing device comprising:

14 a processor that generates a first address signal having a first width;

15 one or more peripheral devices that are addressed with a second address
16 signal having a second width that is greater than the first width;

17 an address extension register that stores an address extension, wherein the
18 address extension is combined with the first address signal to create the second
19 address signal;

20 a multi-tasking operating system that switches between execution of
21 different application programs;

22 a plurality of application programs that use different address extensions,
23 wherein execution threads of the application programs register one or more such
24 address extensions with the multi-tasking operating system;

1 wherein the operating system records the registered address extensions and
2 automatically writes an address extension of a particular application program
3 thread to the address extension register before switching to execution of said
4 particular application program thread.

5
6 **21.** A computing device as recited in claim 20, wherein each address
7 extension identifies an address range associated with one or more peripheral
8 devices.

9
10 **22.** A computing device as recited in claim 20, wherein operating
11 system records the registered address extensions in a memory table.

12
13 **23.** A computing device as recited in claim 20, wherein the execution
14 threads of the application programs invoke operating system device drivers, the
15 operating system device drivers registering said one or more address extensions
16 with the multi-tasking operating system.

17
18 **24.** A computer program stored in a storage medium, comprising:
19 instructions for performing read/write operations on a peripheral device,
20 wherein loading an extension register is a predicate to performing said read/write
21 operations; and

22 instructions for providing one or more address extension values to a multi-
23 tasking operating system for use with one or more threads of the computer
24 program, wherein the operating system automatically loads said one or more
25

1 address extension values to the extension register whenever switching to said one
2 or more threads of the computer program.

3
4 **25.** A computer program as recited in claim 24, wherein the address
5 extensions are specified as base addresses.

6
7 **26.** A computer program stored in a storage medium for execution on a
8 computer, the computer program the following steps:

9 executing an interruptible execution thread of the program;

10 writing an address extension value associated with the execution thread to
11 an extension register and contemporaneously to a memory location;

12 associating the memory location with the execution thread;

13 retrieving the value associated with the execution thread from the memory
14 location when execution of the execution thread is resumed after being
15 interrupted;

16 writing the value retrieved from the memory location to the extension
17 register; and

18 resuming application of the execution thread.

19
20 **27.** A computer program as recited in claim 26, wherein each address
21 extension identifies an address range associated with one or more peripheral
22 devices.

1 **28.** A computer program as recited in claim 26, further comprising a
2 step of alternately executing more than one execution thread.

3
4 **29.** A computer program as recited in claim 28, further comprising a
5 step of storing more than one address extension value in memory, each value
6 being associated with a particular execution thread.

7
8 **30.** A computer program as recited in claim 28, further comprising:
9 storing more than one address extension value in memory, each value being
10 associated with a particular execution thread; and
11 loading the extension register with the value in memory associated with a
12 particular execution thread prior to resuming execution of that execution thread.

13
14 **31.** A computer program as recited in claim 28, further comprising:
15 storing more than one address extension value in memory, each value being
16 associated with a particular execution thread; and
17 loading the extension register with the value in memory associated with a
18 particular execution thread prior to resuming execution of that execution thread;
19 wherein each address extension identifies an address range associated with
20 one or more peripheral devices.

1 32. A computer program as recited in claim 28, wherein the computer
2 program executes an operating system device driver from an interruptible
3 execution thread of the program wherein said device driver identifies a peripheral
4 device to be accessed by the execution thread and identifies a value associated
5 with the execution thread.
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25

1 **ABSTRACT**

2 A multi-tasking operating system and method updates PCI address values
3 in an extension register to ensure that various threads utilize the correct values
4 when accessing peripheral PCI devices. When application program threads
5 require access to a PCI device, the operating system writes the high order bits of
6 the PCI device address to two places: (1) the extension register of the PCI host
7 bridge to allow immediate addressing of the PCI device, and (2) separate memory
8 locations associated with the threads. When a context switch occurs from a first
9 thread to a second thread, the operating system retrieves the stored value from the
10 memory location associated with the second thread and writes the value to the
11 extension register. In this manner, when the second thread requires access to its
12 PCI device, the proper address value is already located in the extension register.

13
14
15
16
17
18
19
20
21
22
23
24
25

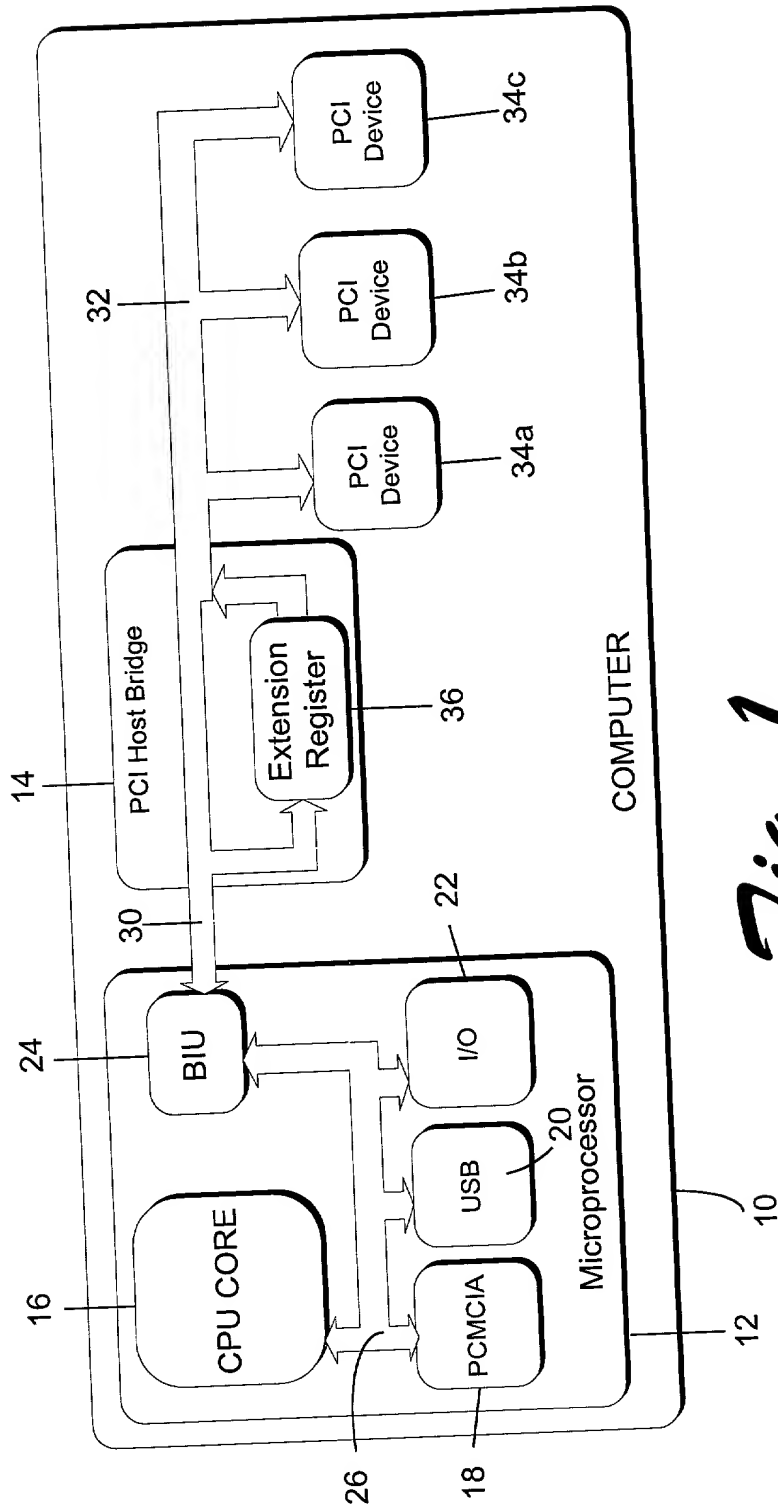
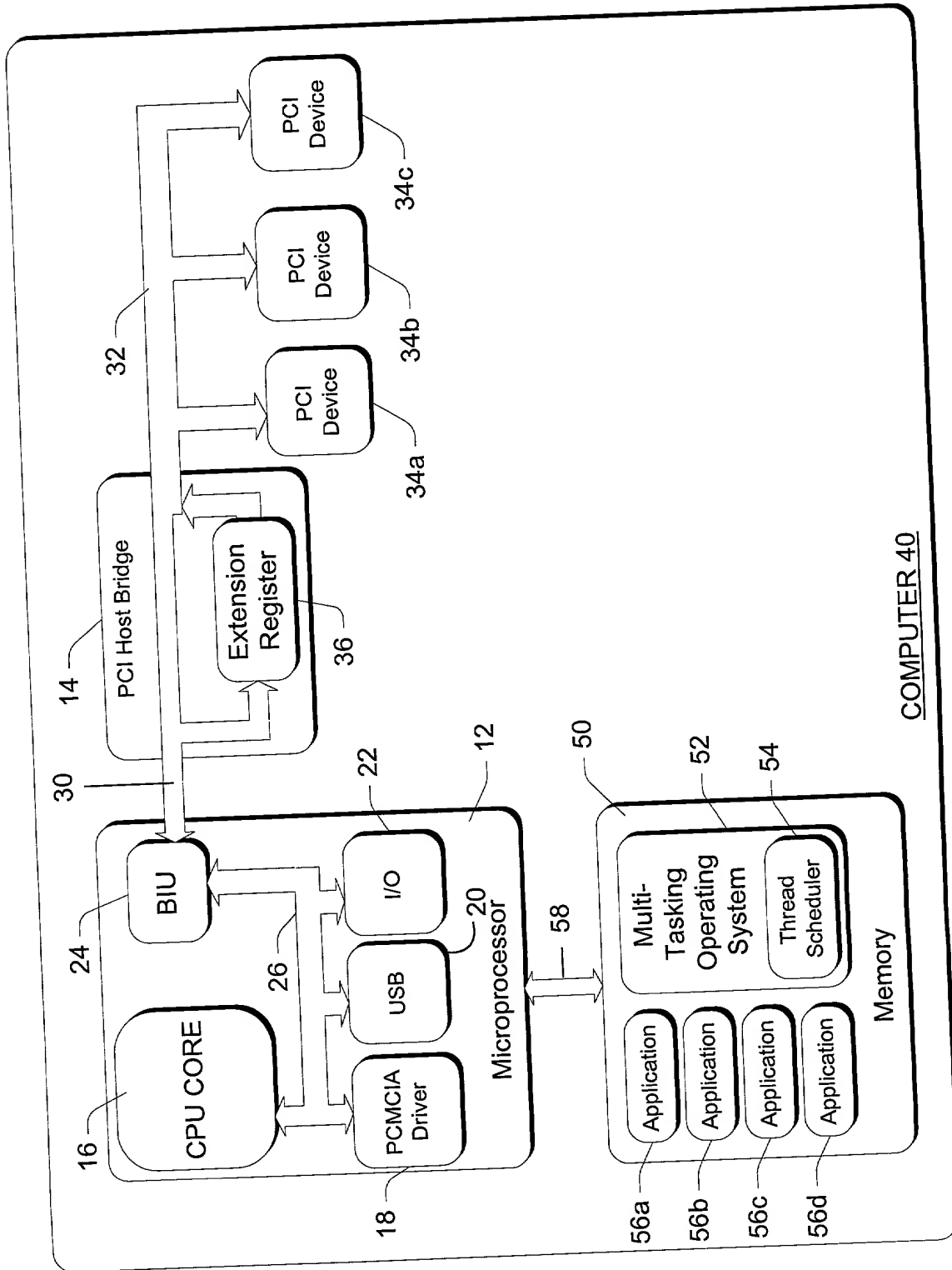


Fig. 1
Prior Art

**Fig. 2**

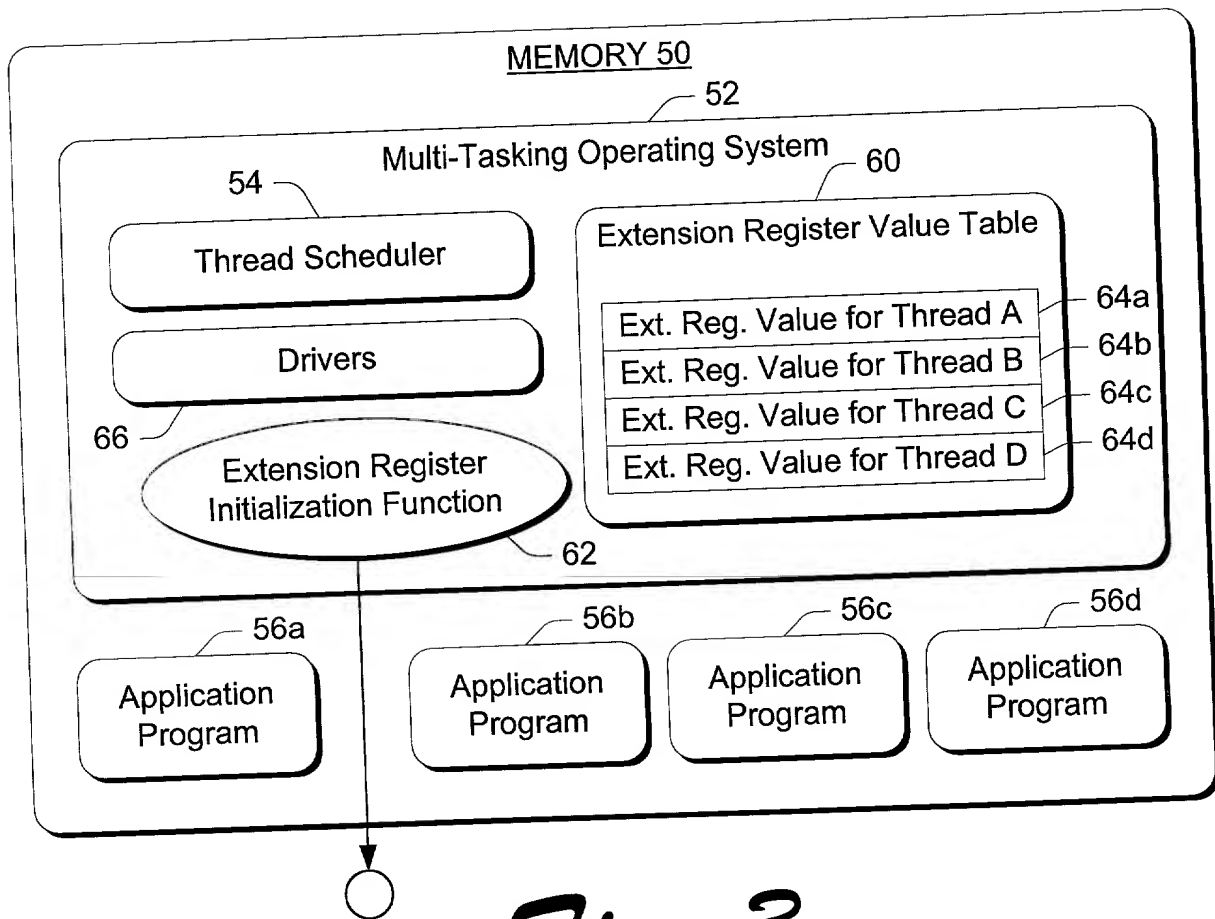


Fig. 3

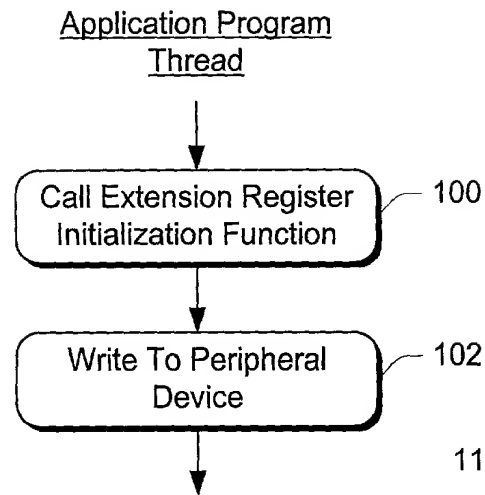


Fig. 4

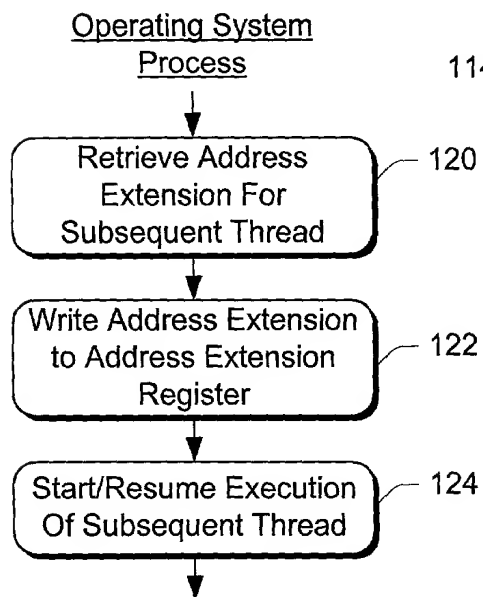


Fig. 6

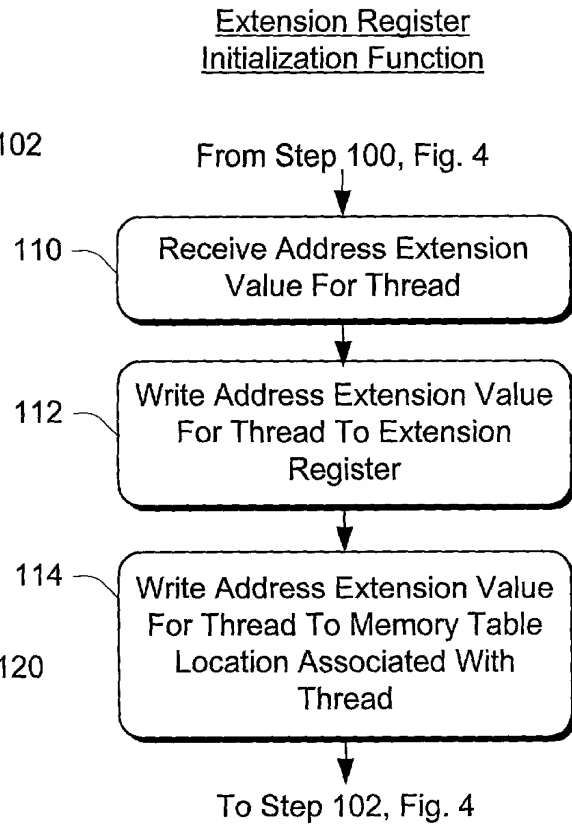


Fig. 5